Towards Architecture Independent Metrics for Multicore Performance Analysis

Milind Kulkarni, Vijay Pai, and Derek Schuff
School of Electrical and Computer Engineering
Purdue University
{milind, vpai, dschuff}@purdue.edu

ABSTRACT

The prevalence of multicore architectures has made the performance analysis of multithreaded applications an intriguing area of inquiry. An understanding of locality effects and communication behavior can provide programmers with valuable information about performance bottlenecks and opportunities for optimization. Unfortunately, most performance analyses are *architecture dependent*, and hence insights gleaned from an application's behavior on one platform may not apply when the application is run on another. In this position paper, we argue that what is needed are *architecture independent* metrics that characterize the behavior of an application in a system-agnostic manner. Such metrics will allow a program's performance to be analyzed across a range of architectures without incurring the overhead of repeated profiling and analysis. We propose two specific analyses: *multicore-aware reuse distance*, which captures the locality properties of an application and *communication analysis*, which exposes the structure of communication in an application. We also discuss a number of applications of these analyses, in the domains of optimization, code restructuring and performance modeling.

1. INTRODUCTION

Performance tuning for multicore systems typically requires manual restructuring of code and new compiler optimizations to target different architectures. These steps are often performed in an ad-hoc fashion, as different applications and different platforms require different forms of tuning. In general, however, performance tuning requires improving data cache locality and reducing communication bottlenecks. Although the performance impacts of locality and communication are architecture-specific, both properties are determined largely by fundamental characteristics of the program such as data reuse and inter-thread interactions. Thus, effective tuning for rapidly evolving architectures will benefit from architecture-independent metrics that capture data reuse and inter-thread interactions in parallel programs.

Reuse distance has long been an architecture-independent quantitative metric for data reuse in programs [12]. The reuse distance of a given reference to element x is the count of the number of distinct data elements that have been referenced since the last access of x (or ∞ if the element has not been referenced before). The data elements can be pages, cache lines, individual words, or even instructions, yielding a generalized machine-independent measure of program locality. Conceptually, this is implemented by tracking the depth of the access in a stack, though more efficient implementations also exist [15]. Reuse distance predicts cache hit ratio; in particular, the hit ratio of a cache with N one-word blocks organized in a single LRU set would correspond to the fraction of references with reuse distance less than N. A single run of reuse distance analysis can thus predict the behavior of an application over a variety of cache sizes, but does not account for real cache constraints like associativity replacement policy details. When considering multicore systems, however, existing reuse distance analysis methods are insufficient, as their abstract notion of access ordering by counting only makes sense within a single reference stream. Recent studies have accounted for multicore effects on reuse distance for systems with shared caches or private coherent caches. Shared caches can be modeled by measuring or statistically estimating the impact of interleaving instructions from multiple threads onto a single reuse stack, thereby accounting for shared-cache effects such as inter-core prefetching, more effective capacity utilization, and capacity contention [6, 7, 13, 14]. Private caches have been modeled by eliminating entries from other threads' stacks when they are written by one thread, mimicking the behavior of invalidation-based coherence protocols [13, 14].

Data communication between threads is another key factor in shaping multithreaded application performance. The performance impact of such communication often depends on the size and sharing configuration of the cache. Numerous works have investigated sharing behavior in applications, but often only for a single cache size [5]. Such works can help understand issues related to sharing, but do not indicate how changes in cache size, configuration, or even thread allocation policies might differently affect performance. A more detailed analysis would additionally consider the different source and destination threads involved in a communication event, the sharing patterns involved, the extent to which addresses in communication patterns are correlated over time, the number of concurrent communication events in the system, and the extent to which communications correlate to

particular regions of the source code. Such analysis would focus on inter-thread interactions rather than the specific workings of communication for a given platform, and could thus yield insights in an architecture-independent sense of how program code and resource utilization should be optimized to yield high performance across architectures and to properly exploit the tradeoff that sometimes arises between concurrency and data traffic.

Combining multicore-aware reuse distance analysis and communication analysis can help provide understanding about locality, bandwidth, contention, and scalability limitations in multithreaded programs without being tied to the specific architectural details. Despite these benefits, these analyses will not be useful unless they can model full-sized applications with no more than about an order of magnitude slowdown. Random sampling and parallelized analysis are key mechanisms to achieve these goals. Random sampling of individual references is not straightforward for either reuse distance analysis or communication analysis because each analysis depends on substantial state information that has accumulated during the execution of the program. Similarly, parallelization is challenging because both mechanisms rely on shared state. (Both of these issues arise for sampling and parallelization of architectural simulation as well.) This work presents solutions to these challenges along with preliminary work on architecture-independent analyses for multicore shared-memory systems. In the case of reuse distance, the resulting strategy uses sampling not only to avoid tracking all references, but also to enable the use of more efficient data structures and to facilitate parallelization.

There have been other related efforts in a variety of areas. Lee and Brooks used statistical approaches rather than only detailed architectural simulation to identify desired microarchitectural features [9]. In stack reuse distance modeling, works both by Jiang et al. and by Chilimbi and Ding use statistical models to extend per-thread stack reuse distance measures to estimate the behavior of shared-cache systems [7, 6]. Sampling has also been studied for "time" reuse distance analysis (while stack reuse distance counts distinct addresses accessed between two references to the same address, time reuse distance only measures the total number of accesses, whether distinct or repeated) [1, 2]. Our approach differs from previous efforts by using statistical sampling to measure actual stack reuse distances for shared-memory multicore systems, and also by integrating reuse distance analysis with communication analysis to target scalability concerns more directly.

2. MULTICORE REUSE DISTANCE ANALYSIS

Reuse distance is a popular architecture-independent metric that measures data locality in programs; a single run of reuse distance analysis suffices to model all possible memory hierarchies. It operates by using a stack to track the depth of accessed memory addresses in LRU order [12]. However, in its traditional form, reuse distance analysis only considers references from a single stream. This is not sufficient to model the behavior of multithreaded programs, as it ignores the possibility of interference caused by other threads; this interference can impact, or even dominate, the performance of a parallel program. The nature of this inter-core interference is largely dependent on whether cores share caches or have private caches.

Consider the behavior of a multicore system with private caches. Reuse distances are a measure of locality, with shorter distances being more likely to represent cache hits, and longer distances less so. However, if a thread accesses a location and then a second thread writes the same location, the first thread will experience a miss when it next accesses the location, irrespective of the reuse distance. Similarly, if two threads share a cache, then a thread can experience a cache hit even if it never accessed a location before, provided the other thread accessed the location earlier.

Schuff et al. present a multicore-aware model of reuse distance analysis that accounts for these inter-thread interactions during measurement [14]. For shared cache modeling, all threads share a single reuse stack in a straightforward manner. Modeling private caches is more subtle. Each thread has its own reuse stack. In a real cache, when a block is evicted due to an invalidation, it leaves a "hole" that can be filled in by the next block fetched into the cache; similarly, each thread tracks the holes in its private reuse stack left by invalidations. If a block from below the hole (or a never-before-seen block) is accessed, the hole is filled in. A new hole is placed at the prior location of the accessed block. This ensures that blocks do not decrease in depth unless they are accessed. If the hole is filled in by a newly-accessed block, the hole can be removed.

These approaches to handling private and shared caches effectively model both the constructive and destructive interference experienced by multithreaded programs at the cost of analysis efficiency. In general, reuse distance analysis is a high overhead analysis technique; the problem is exacerbated due to the inter-thread interactions modeled by the multicore-aware analysis. In fact, this close coupling precludes the exploitation of parallelism.

2.1 Sampled, parallel analysis

One possible solution to the problem of analysis performance is to perform re-use distance *sampling*. Rather than tracking the reuse distance of every access, the analysis will instead randomly sample addresses from the reference stream and collect reuse distances only for the sampled addresses. If the sampling is performed infrequently enough, the majority of the analysis will be spent waiting for the next sample to be selected (at low overhead)—this is the *fast* mode of operation. When a sample is selected, each thread need merely count how many accesses to distinct addresses are performed until that address is reused (allowing the use of hash-sets rather than stacks to track the reuse distance)—this is the *analysis* mode.

An added attraction of the sampled analysis is that it lends itself to parallelization. While the analysis is in fast mode, each thread is simply counting references until the next sample. As this is a purely local operation, this mode can be readily parallelized, and its performance is constrained only by the parallelism of the profiled application. While in analysis mode, parallelism can be obtained because threads only need to synchronize when a sampled address is reused or an address accessed by one thread is invalidated by another. We can gain additional parallelism by exploiting a relaxed consistency model: reuse distances are not significantly affected by delaying the propagation of an invalidation until the next synchronization point [13], reducing the frequency of synchronization between threads.

Table 1: Accuracy, slowdown (relative to uninstrumented code), and fraction of references analyzed for sampled histograms

Benchmark	Samples	Acc.	Slwdn.	%Refs
applu	141814	98.6%	20.2	8.1%
CG	2335	95.9%	26.8	22.9%
ferret	9804	97.1%	24.5	6.8%

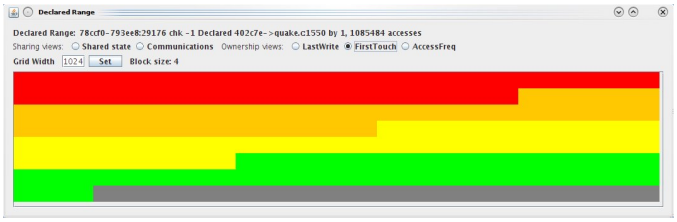
We implemented both the full multicore reuse distance analysis and a parallel, sampled analysis, using PIN [10] to instrument our target applications. Table 1 shows the accuracy of the sampled analysis compared to the full analysis for a selection of benchmarks from the NAS, SpecOMP and Parsec suites. The sampled analysis took one sample every one million references, on average. The accuracy represents the similarity of the reuse distance histograms generated by the sampled and full analyses. This is determined by normalizing the histograms from full and sampled analyses to make each histogram bin represent the fraction of references in the given range of reuse distances, and then summing up the absolute difference between the two histograms at each bin to get a cumulative error value from 0% to 200% (called E). The accuracy is then computed as $1 - \frac{E}{2}$. The slowdown column gives the slowdown of the sampled analysis relative to the uninstrumented application. %Refs gives the percentage of references seen by the sampled analysis while in analysis mode. We see that the sampled analysis is both highly accurate and reasonably fast. Detailed analysis shows that sampling analysis is (on the average) 177x faster than full analysis, with benefits from the use of fast mode, intelligent parallelization (e.g., synchronization reduction, privatization of data structures), and because sampling enables the use of hash sets rather than trees for analysis.

3. COMMUNICATION ANALYSIS

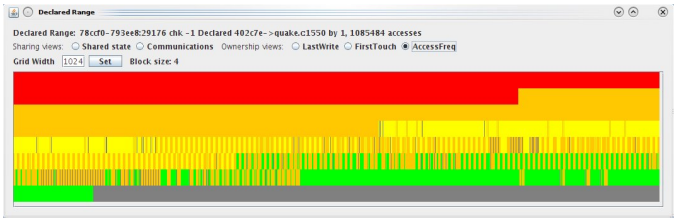
Although the details of communication performance bottlenecks are typically considered to be intimately connected to the implementations of specific architectures and interconnects, communication itself most often arises from interactions between different threads. Thus, architecture-independent analysis of a program execution can also highlight inter-thread interactions as sources of potential communication bottlenecks.

To study these interactions, we have designed a tool that processes an execution trace generated by Simics or Pin and tracks data addresses that are touched by more than one thread. Every load or store is monitored according to the thread that triggered it; additionally, every store of an address by one thread followed (at any point in the future execution) by a load or store of that same address by some other thread is marked as a communication event. Every data memory reference is thus turned into a 5-tuple with the following information: data address, thread, PC, source thread, source PC. The source fields are only relevant on communication events.

The tool then allows various visualizations of the communication pattern. Figure 1 shows how data is partitioned among threads in the SpecOMP benchmark `equake`. Figure 1(a) splits a data region among threads (shown in diferent colors) based on first-touch. In contrast, Figure 1(b) partitions the same data according to access frequency: the thread that accesses a given address most frequently is said



(a) Partitioning by first touch



(b) Partitioning by access frequency

Figure 1: Data partitioning among 4 threads in SpecOMP benchmark `quake`

to be the owner of that piece of data. While first-touch splits the data evenly and predictably, it is not consistent with the actual data access frequency. First-touch is commonly used in NUMA systems for data allocation, with some systems supporting page-level migration to account for variations during execution [8]. Such coarse-grained migration, however, will not compensate for the fine-grained variations in access frequency implied by these figures.

3.1 Sampled communication analysis

The full communication analysis tool relies on memory traces from every thread. These logs can be very large, and processing them to generate communication profiles is time consuming. As a result, applying communication analysis tool to larger data sets is infeasible. Unfortunately, unlike reuse distance analysis, it is unclear how to apply sampling techniques to communication analysis. Communication analysis relies on turning a stream of memory references into a stream of tuples that associates each memory reference with the previous "owner" of the memory location. It is apparent that sampling from the *tuple stream* can provide useful communication information. Unfortunately, the chief cost of communication analysis is in generating the tuple stream from memory traces; rather than applying sampling at this downstream point in the analysis, we must apply it earlier.

There are several possible sampling methods that may be profitable. One option is to sample in *space* rather than time. Such an analysis would sample from the accessed memory locations (rather than sampling from memory references) and track *all* accesses to those locations. This may produce accurate results under the assumption that nearby memory locations have similar communication characteristics. An alternate approach is to use "on-off" sampling, where the full analysis is run during sampled intervals of the program. This allows us to precisely capture communication during the period we analyze at the cost of losing all information during other periods; provided the program runs for long enough, and our choice of "on" periods is unbiased, we may still be able to generate accurate results. As a final alternative, we can use static analysis to (conservatively) identify

which addresses are involved in communication, and apply the analysis only to references to those addresses, ignoring the remainder.

4. APPLICATIONS

4.1 Reuse distance analysis

4.1.1 PC selection

One application of reuse distance analysis is determining which portions of a program contribute most to poor locality and assisting the programmer in making improvements [2, 4, 11]. This requires identifying which PCs are responsible for the bulk of misses suffered by a program.

Suppose we would like to concentrate our optimization effort on a small number of PCs. For example, the top N PCs that are responsible for the misses at a particular cache size C. This can be easily achieved by using multicore reuse distance analysis: we perform a single run of the analysis, and for each memory access record both the stack depth of the reuse and the PC that performed the access. We can then select only the PCs that incur reuses of depth greater than C (and hence would be misses if the application were executed on a machine with caches of size C). By ranking the PCs according to the number of predicted misses they cause, we can select the top PCs that are responsible for the bulk of misses in a program, narrowing the focus of our optimization efforts.

This simple version of PC selection can be extended to provide additional information that programmers can use when optimizing their programs. For example, by including information about which code is executed between a use and its reuse, an analysis can guide refactoring [3]. This information can easily be recorded during a run of multicore reuse distance analysis.

4.1.2 Modeling hierarchies of private/shared caches

A natural extension of multicore-aware reuse distance analysis is to model the behavior of modern cache hierarchies, which often consist of private L1 caches and shared L2 caches. This means that short reuse-distance accesses should be treated as if their data reside in private caches, while longer-distance accesses should be modeled as if they are manipulating a shared cache. Unfortunately, the threshold at which an access has a "short" reuse distance versus a "long" reuse distance is dependent on the size of the L1 cache, and hence entails architecture dependence.

However, it may be possible to simultaneously model all possible private/shared cache hierarchies. This can be done by performing both a private-cache analysis and a shared-cache analysis concurrently during a profiling run. To determine the miss rate for a particular cache hierarchy, the results of the private cache run are consulted first to determine if a particular access misses in the private, L1 cache. If it does, then the stack distance of the access in the shared-cache profile is used to determine if the access misses in the shared, L2 cache.

While this approach does not directly model the behavior of a private/shared hierarchy, it will accurately determine the miss rate in the L1 (up to the accuracy of the underlying private-cache analysis). The miss rate in the L2 may not be accurately modeled due to the differences between inclusive and exclusive L2 caches, and the exact operation of the L2 when an address is evicted from the L1. However, because the L1 cache is typically much smaller than the L2, these effects should be minimal, and the miss rate estimate should be reasonable.

4.2 Communication analysis

Communication analysis is useful in distributed-memory environments such as distributed shared memory and MPI, because its definition as data produced by one thread and consumed by another thread places a lower bound on the amount of data that must be transmitted between threads. In systems where the memory is truly shared this may not matter because the cost of accessing data (e.g. in main memory) may be the same regardless of which thread has written it; however, with distributed memory the data must always at least be transmitted from the writer to the reader, either explicitly (as in MPI) or implicitly (as in software DSM). A program's communication behavior then becomes a useful indication of performance or resource usage in such an environment.

4.3 Combined

As standalone analyses, both communication analysis and multicore-aware reuse distance analysis are useful tools for optimizing programs and architectures. However, they target different aspects of the locality and communication problem: reuse distance analysis tells a programmer which *data* is likely to trigger cache misses, while communication analysis tells a programmer which *thread* is likely to trigger communication. To fully understand the behavior of a program, it is necessary to combine this information. For example, if communication analysis identifies several threads as communicating amongst one another, reuse distance analysis can identify whether this communication is a problem or not. If the communication triggers frequent invalidations, then it should be addressed; but if the memory locations have long reuse distances, they would rarely be in cache and the communication would not trigger invalidations. By combining the two analyses, we can provide an architecture-independent analysis that allows programmers to gain deeper insight into the locality, sharing, and communication properties of their programs than using either analysis alone.

4.3.1 Modeling memory systems

An important factor in the performance of an application is the behavior of the memory system, as excessive cache misses increase latency and cause bandwidth saturation. Bandwidth is not only shared between cores accessing main memory, it is also needed for cache-to-cache data transfers. Because cache misses and data traffic have a significant impact on performance, it is crucial to be able to model these behaviors.

Currently, the only ways to study application memory system performance are through the use of performance counters, which can track cache misses and bus transactions, or through simulation. Unfortunately, both approaches are tied to a specific architecture and provide little insight into the behavior of other memory system configurations. We propose a two-level modeling approach. First, we will characterize an application according to its inherent, architecture-independent behavior. Second, we will feed this information to architecture-specific memory system models, predicting

the behavior of the application on various architectures without running the application on each system.

Multicore-aware reuse distance analysis can predict the miss ratio for any cache size, including systems with private L1 caches and shared last-level caches. However, we must also account for the effects of cache-to-cache transfers. Unfortunately, reuse distance analysis does not provide sufficient information to estimate the incidence of cache-to-cache transfers. Communication analysis exposes the communication behavior of threads, but an abstract message sent between two threads does not necessitate a cache-to-cache transfer. Combining the two analyses provides the solution. If we record the current stack distance of a location that is communicated between two threads, we can determine if the communication triggers a cache-to-cache transfer (which will be the case if the stack distance in the source thread is shallow enough so that the data will be in cache) or if it merely requires access to main memory. Note that even though determining the effects of communication requires knowledge of the memory system organization, the *data* needed for that decision (stack distances and communication patterns) is inherent to the application, and not dependent on the architecture. Combining multicore-aware reuse distance analysis and communication analysis allows a single architecture-independent run to collect data that can be used to predict both cache misses and bandwidth utilization for any memory system.

5. CONCLUSIONS

Architecture-independent performance metrics allow a program to be characterized and studied without appealing to a particular hardware configuration. We propose two such analyses to capture the behavior of multithreaded programs. Multicore-aware reuse distance analysis provides a measure of locality that can be used to predict cache behavior, even in the presence of constructive and destructive cache effects, while communication analysis exposes the sharing and communication behavior of threads in a shared-memory program. We argue that these metrics can be used to enable a number of optimization and performance analysis tasks.

Ultimately, architecture-independent metrics for parallel programs enable an attractive approach to performance modeling and analysis. Rather than analyzing the behavior of a particular program on a particular architecture, programs can be analyzed once using architecture-independent metrics. The results of these metrics can then be used by architecture-specific performance models to predict a program's behavior. By separating the collection of metrics from the development of performance models for an architecture, the amount of time required to analyze the behavior of multiple programs across multiple architectures can be significantly reduced.

6. REFERENCES

[1] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. *2006 IEEE International Symposium on IEEE Performance Analysis of Systems and Software*, pages 89–99, March 2006.

[2] K. Beyls and E. D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In *4th International Conference on Computational Science*, pages 448–455, 2004.

[3] K. Beyls and E. D'Hollander. Discovery of locality-improving refactorings by reuse path analysis. In *Proc.2nd International Conference on High Performance Computing and Communications (HPCC)*, 2006.

[4] K. Beyls, E. D'Hollander, and F. Vandeputte. Rdvis: A tool that visualizes the causes of low locality and hints program optimizations. In *5th International Conference on Computational Science*, 2005.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[6] C. Ding and T. Chilimbi. A composable model for analyzing locality of multi-threaded programs. Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[7] Y. Jiang, E. Zhang, and K. Tian. Is reuse distance applicable to data locality analysis on chip multiprocessors. In *International Conference on Compiler Construction*, Paphos, Cyprus, March 2010.

[8] J. Laudon and D. Lenoski. The SGI Origin 2000: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[9] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 185–194, 2006.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[11] G. Marin and J. Mellor-Crummey. Pinpointing and exploiting opportunities for enhancing data reuse. In *2008 IEEE International Symposium on Performance Analysis of Systems and Software*, 2008.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[13] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. Technical Report TR-ECE-09-08, Purdue University, 2009.

[14] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. In *Workshop on Performance Modeling, Evaluation, and Optimisation of Ubiquitous Computing and Networked Systems*, 2010.

[15] R. A. Sugumar and S. G. Abraham. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical report, University of Michigan, 1993.